LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-675389

KRIPKE - A MASSIVELY PARALLEL TRANSPORT MINI-APP

A. J. Kunen, T. S. Bailey, P. N. Brown

July 29, 2015

American Nuclear Society Joint International Conference on Mathematics and Computation, Supercomputing in Nuclear Applications, and the Monte Carlo Method
Nashville, TN, United States
April 19, 2015 through April 23, 2015

Disclaimer

ANS MC2015 - Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method · Nashville, Tennessee · April 19–23, 2015, on CD-ROM, American Nuclear Society, LaGrange Park, IL (2015)

KRIPKE - A MASSIVELY PARALLEL TRANSPORT MINI-APP

Adam J. Kunen, Teresa S. Bailey, and Peter N. Brown
Lawrence Livermore National Laboratory
Livermore, California 94551
kunen1@llnl.gov, bailey42@llnl.gov, pnbrown@llnl.gov

ABSTRACT

As computer architectures become more complex, developing high performance computing codes becomes more challenging. Processors are getting more cores, which tend to be simpler and support multiple hardware threads, and ever wider SIMD (vector) units. Memory is becoming more hierarchical with more diverse bandwidths and latencies. GPU's push these trends to an extreme. Existing simulation codes that had good performance on the previous generation of computers will most likely not perform as well on new architectures. Rewriting existing codes from scratch is a monumental task. Refactoring existing codes is often more tractable. Proxy Applications are proving to be valuable research tools that help explore the best approaches to use in existing codes. They provide a much smaller code that can be refactored or rewritten at little cost, but provide insight into how the parent code would behave with a similar (but much more expensive) refactoring effort. In this paper we introduce KRIPKE, a mini-app developed at Lawrence Livermore National Laboratory, designed to be a proxy for a fully functional discrete-ordinates (S_N) transport code. KRIPKE was developed to study the performance characteristics of data layouts, programming models, and sweep algorithms. KRIPKE is designed to support different in-memory data layouts, and allows work to be grouped into sets in order to expose more on-node parallelism. Different data layouts change the way in which software is implemented, how that software is compiled for a given architecture, and how that generated code eventually performs on a given architecture.

Key Words: Mini-App, Boltzmann, S_N transport, sweeps, scaling

1 INTRODUCTION

As computer architectures become more complex, developing high performance computing codes becomes more challenging. Processors are getting more cores, which tend to be simpler and support multiple hardware threads, and ever wider SIMD (vector) units. Memory is becoming more hierarchical with more diverse bandwidths and latencies. GPU architectures push these trends to an extreme, requiring tens of thousands of threads to make full use of the hardware. Existing simulation codes that had good performance on the previous generation of computers will most likely not perform as well on new architectures. Big shifts in algorithmic and/or software engineering approaches are often required to achieve reasonable performance on these new platforms. [1]

Rewriting existing codes from scratch is a monumental task. Refactoring existing codes is often more tractable. Due to the uncertainty about which algorithmic and software approaches are best for current or future architectures, there is a large risk associated with any major refactoring

effort. Proxy Applications are proving to be valuable research tools that help us explore the best approaches to use in our existing codes. They provide a much smaller code that can be refactored or rewritten at little cost, but provide insight into how the parent code would behave with a similar (but much more expensive) refactoring effort.

In this paper we introduce KRIPKE [2], a mini-app developed at Lawrence Livermore National Laboratory, designed to be a proxy for a discrete-ordinates (S_N) transport code called ARDRA [3,4]. KRIPKE was developed to study the performance characteristics of data layouts, programming models, and sweep algorithms. Unlike ARDRA, KRIPKE is designed to support different in-memory data layouts, and allows work to be grouped into sets in order to expose more on-node parallelism. Different data layouts change the way in which software is implemented, how that software is compiled for a given architecture, and how that generated code performs on a given architecture. KRIPKE is intended to have the same scaling performance characteristics as ARDRA, so we can study parallel decompositions and algorithms, as well as how the on-node performance choices affect large scale performance.

First we describe the transport problem under consideration. Then we discuss some general implementation issues involved with writing an S_N transport code. We provide a detailed discussion about implementation and design choices made during the development of KRIPKE. Next, we provide some experimental results that demonstrate KRIPKE's success as a proxy for ARDRA. We also include computational results that demonstrate KRIPKE's usefulness as a research tool. We then make some concluding remarks.

2 PROBLEM DESCRIPTION

We consider the steady state form of the Boltzmann transport equation in three dimensional geometry, with the angular flux $\psi(r, \Omega, E)$ the solution to

$$[\Omega \cdot \nabla + \sigma(r, E)]\, \psi(r, \Omega, E) = \int dE' \int d\Omega' \sigma_s(r, \Omega' \cdot \Omega, E' \rightarrow E)\psi(r, \Omega', E') + q(r, \Omega, E), \quad (1)$$

where σ and σ_s are the total and scattering cross sections, respectively, q is a source, and ∇ is the spatial gradient. The spatial domain is $\mathcal{D} \subset \mathbf{R}^3$, with $r \in \mathcal{D}$. The direction (or angle) variable is $\Omega \in \mathcal{S}^2$, the unit sphere in $\mathbf{R}^3$, and the energy variable is $E \in (0, \infty)$. Thus the angular intensity, ψ, resides in a six-dimensional phase space. See [5] for more details.

Using a standard discrete-ordinates discretization of (1), the energy variable E is binned into G intervals or groups, the integral over Ω is accomplished using a quadrature rule with D points and weights, and the spatial operator is discretized using a Diamond Difference approximation with Z zones. Thus, the number of unknowns in a discrete problem is $G \cdot D \cdot Z$. The discretized form of the transport problem (1) can be written in matrix form as

$$H\Psi = L^{+}\Sigma_s L\Psi + Q, \quad (2)$$

where Ψ is the discrete flux, H is the discrete form of the streaming plus collision operator $\Omega \cdot \nabla + \sigma$, Q is the discrete source, L is the discrete moment operator derived from the discrete-ordinates

quadrature approximating the integrals $\int \psi(r, \Omega, E) Y_n^m(\Omega) d\Omega$, where $|m| \leq n$, $n = 0, \cdots, N_s$, the scattering order, and $Y_n^m(\Omega)$ the standard spherical harmonics. The variable $\Phi = L\Psi$ is called a moment vector, containing the moments of Ψ. The matrix L^+ is a right inverse of L, so that LL^+ is the corresponding identity matrix. Thus, $L^+\Phi$ is the same size as the unknown Ψ. The matrix Σ_s represents the discretized scattering operater. See [6] for a more detailed description of the discrete problem.

Because of the six-dimensional phase space, reasonably resolved simulations require high unknown counts, so iterative solution techniques are normally used. Many iterative methods can be applied to solve the system in (2), [7]. Scaling to high processor counts requires not only scaling of the algorithm that executes a single iteration, but also an iterative method whose iteration counts do not grow with resolution. An example iterative solution approach for (2) has the form

- Choose an initial guess Ψ^0.
- For $i = 0, 1, \cdots$ until convergence, solve

$$H\Psi^{i+1} = L^+\Sigma_s L\Psi^i + Q. \tag{3}$$

At the core of this iteration is what we define as the "sweep", the inversion of the matrix H given the right hand side in (3). For the iteration in (3), note that the matrix H is block-diagonal in groups and directions, and the iteration can be described as a Block-Jacobi in energy method. All scattering from group to group is handled when computing the right hand side of (3). Other approaches in energy include a Gauss-Seidel iteration, wherein the lower-triangular part of the scattering matrix is moved to the left hand side of (3) [8]. For example, we could write $L^+\Sigma_s L = \hat{L} + \hat{U}$, with $\hat{L}$ block lower-triangular and $\hat{U}$ strictly block upper-triangular in energy. Then (3) can be rewritten as

$$(H + \hat{L})\Psi^{i+1} = \hat{U}\Psi^i + Q.$$

In this case, there is group-to-group coupling when inverting $H + \hat{L}$. In either case, performing a sweep involves following "wavefronts"—one per quadrature direction—across the spatial domain, for each energy group. Algorithms that rely on sweeps generally have the property that the iteration count does not increase with increasing spatial or directional resolution. We employ a mini-app described below to investigate the scalability and performance of sweep-based transport algorithms for advanced architectures.

3 IMPLEMENTATION ISSUES

The computational efficiency of a discrete-ordinates transport solver depends on how well the code exploits the computer's hardware resources and takes advantage of hierarchical parallelism. Modern CPU's require the use of SIMD vector units to make full use of their floating point hardware. GPU's, and many advanced CPU architectures, require the use of many-way threading to achieve full instruction throughput and hide memory latency. Larger problems, which run on multi-node clusters in order to have enough memory and computational resources, require parallel

decomposition of the problem and the use of message passing protocols. This gives a view of a hierarchical parallel programming setting involving instruction level parallelism (SIMD), thread level parallelism (software or hardware threads), and task level parallelism (multi-nodes). In this paper we use the term task to refer to MPI processes. Other hardware resource considerations are multi-level cache or memory architectures and NUMA effects. The way in which data is ordered and decomposed across nodes greatly affects the way it maps to each level of parallelism on a particular hardware architecture, and therefore determines its performance. Design decisions must be made that affect the implementation of the code (software engineering) and performance. [1]

On a parallel computer running P tasks we can partition $G \cdot D \cdot Z$ into P sets, identified by $\{G_p \cdot D_p \cdot Z_p\}_{p \in P}$. We call each p a "subdomain" of the entire problem domain, which map 1-to-1 with each parallel task. Likewise, we can write Ψ_p to represent Ψ in the p^{th} subdomain. For the purpose of this paper we only discuss the parallel partitioning of Z, however codes exist that also allow the partitioning of G and D [4, 8, 9].

At the task level, a data layout decision must be made. The layout of Ψ_p in ARDRA can be thought of as the three dimensional C++ array, $\Psi_p[G_p][D_p][Z_p]$, which maps to a linear address space on the computer. In this case each zone is stride 1, each direction is stride Z_p and each group is stride $G_p \cdot D_p$. Similarly, Φ_p has the same structure with moments replacing directions, $\Phi_p[G_p][M_p][Z_p]$. (Note that in 3D the number of moments depends upon the scattering order N_s, with the total number of moments $M = (N_s + 1)^2$.) However, one can also choose any of the six permutations for this data layout:

$$
\begin{array}{ll}
\Psi_p[D_p][G_p][Z_p], & \Phi_p[M_p][G_p][Z_p] \\
\Psi_p[D_p][Z_p][G_p], & \Phi_p[M_p][Z_p][G_p] \\
\Psi_p[G_p][D_p][Z_p], & \Phi_p[G_p][M_p][Z_p] \\
\Psi_p[G_p][Z_p][D_p], & \Phi_p[G_p][Z_p][M_p] \\
\Psi_p[Z_p][D_p][G_p], & \Phi_p[Z_p][M_p][G_p] \\
\Psi_p[Z_p][G_p][D_p], & \Phi_p[Z_p][G_p][M_p].
\end{array}
$$

We will refer to these data layouts as DGZ, DZG, GDZ, GZD, ZDG, ZGD, respectively, and constrain ourselves to always having the same data layout for Ψ and Φ. When writing any code that acts upon Ψ, one must write nested loops over groups, directions (and/or moments) and zones. To achieve high-performance, we want to access memory as sequentially as possible. This means that the outermost loops have the largest stride, and the innermost loops have the shortest stride. Because the nesting order of these loops change in relation to the data layout, we refer to each of these data layouts as "nestings".

As an example, let us examine what we call the LTimes kernel which computes $\Phi = L\Psi$. Since we are mapping direction space to moment space, we can see that each group and zone are independent. Therefore, LTimes can be written as a block-diagonal matrix-vector multiply with one block for each group-zone pair:

$$\Phi(g, z, m) = \sum_{d \in D} L(d, m)\Psi(g, z, d). \tag{4}$$

If we write this out for the GZD nesting, it would look something like this pseudocode:

```
function LTimes_GZD:
    for (g,z) in (G,Z):
        for m in M:
          Phi[g][z][m] = 0
        end
        for d in D:
            for m in M:
                Phi[g][z][m] += L[d][m] * Psi[g][z][d]
            end
        end
    end
end_function
```

whereas the same kernel with the DZG nesting would look like

```
function LTimes_DZG:
    for (m,z,g) in (M,Z,G):
      Phi[m][z][g] = 0
    end
    for d in D:
        for m in M:
            for (z,g) in (Z,G):
                Phi[m][z][g] += L[d][m] * Psi[d][z][g]
            end
        end
    end
end_function
```

Both of these kernel variants should be able to achieve SIMD vectorization. However the inner-most loop in the GZD variant may be fairly short, as there are only $M = (N_s + 1)^2$ inner loop iterations. In the DZG variant we can collapse the inner-most two loops into one. If the number of elements in a vectorized loop are not divisible by the width of an architecture's SIMD vector unit, the last few elements of a loop will not vectorize. If an architecture requires aligned memory for vector operations, the beginning of a loop may not vectorize. A longer inner-loop will help reduce the relative amount of time in these loop "preambles" and "postambles", and allow for more of the loop to be vectorized [10]. Therefore, we should expect that the DZG variant will be much more efficient for vectorization than the GZD variant.

Applying OpenMP loop-level threading also has its trade-offs. Since each (g, z) is independent for LTimes, threading the outer loop in the GZD variant is straight-forward and requires no critical sections or locking. However, threading the DZG variant is troublesome. Once can either apply loop-level threading to the inner-most loop, or to the outer m or d loops. Threading the inner-most loop is safe and requires no locking but reduces the length of each threads loop, hence reducing the vectorization efficiency mentioned earlier. Threading either of the outer loops introduces large data dependency issues, and a large amount of cache inefficiency.

4 KRIPKE IMPLEMENTATION

The mini-application KRIPKE was developed as a proxy to ARDRA in order to study the performance characteristics of data layouts, programming models, and sweep algorithms. A useful mini-app should be small and portable so that it can be easily understood, rewritten and/or refactored in a small amount of time, allowing a large number of research avenues to be explored with minimal expense. ARDRA is roughly 200 thousand lines of code, whereas KRIPKE is only about 2 thousand lines of code. A good proxy application also should be respresentative of a parent application so that research results from the proxy are useful results for the parent.

KRIPKE, like ARDRA, has been implemented with C++, OpenMP [11] and MPI [12]. We have implemented a simplified steady state solver in which there is no external source term Q, and the scattering matrix is the identity, $\Sigma_s = I$. This gives us for equation (3) in KRIPKE,

$$H\Psi^{i+1} = L^+ I L\Psi^i + 0 = L^+ L\Psi^i. \tag{5}$$

Solving equation (5) only requires three operators to be implemented: the L operator (LTimes), the L^+ operator (LPlusTimes) and the H^{-1} operator (Sweep). KRIPKE's sweep algorithm assumes a structured grid, which is consistent with the current capability in ARDRA. Both sweep algorithms contain flexibility that allows the sweep to scale to large numbers of MPI cores [13]. KRIPKE provides spatial-only decomposition across MPI tasks, where each task contains all of the groups and directions for the local subset of zones (i.e., $D_p = D$ and $G_p = G$ for all p). We will show in our results that these features reflect most of the computational intensity found in ARDRA.

Since zones are independent for LTimes and LPlusTimes, they require no MPI communication and act on all of the local unknowns in one call. The wavefront nature of the Sweep leads to multiple simultaneous wavefront communication patterns originating from each of the eight corners of the problem domain. This is implemented in KRIPKE as a parallel Sweep MPI algorithm that handles communication, and a local SweepKernel that performs the local inversion of H using the Diamond Difference discretization. The MPI algorithm asynchronously sends and receives (with isend and irecv) subdomain boundary information as they become available, and calls the SweepKernel to perform the local work [4]. Only one SweepKernel is called at a time, so all of the thread and instruction level parallelism is exploited within the SweepKernel using OpenMP and SIMD vectorization where applicable. This is basically the same Sweep algorithm implementation that is in ARDRA.

To explore parallel and on-node performance we further decompose our unknowns on each task into GS GroupSets and DS DirectionSets. This decomposes our local unknowns into a 5-dimensional vector $\Psi_p[GS][DS][\bar{G}][\bar{D}][Z_p]$, where every task has the same number of GroupSets and DirectionSets, where $\bar{G}$ is the number of groups per set, with the total number of groups being $G = GS \cdot \bar{G}$, and similarly $D = DS \cdot \bar{D}$ for directions. The GroupSet and DirectionSet concepts come from PDT, a code being developed at Texas A&M University [14].

GroupSet's are swept sequentially and DirectionSet's are pipe-lined through the MPI Sweep [13]. In other words, we perform a full parallel sweep of all DirectionSet's for one GroupSet before

moving on to the next GroupSet. The SweepKernel acts upon the unknowns in one GroupSet and DirectionSet at a time, and therefore the "unit of work" done at each stage in the parallel algorithm is over $\bar{G} \cdot \bar{D} \cdot Z_p$ unknowns. This is an extension from ARDRA, which can be thought of as having a fixed $\bar{G} = \bar{D} = 1$.

In order to explore the effect of data layout on performance we implemented all six nestings. Therefore the unknowns $\Psi_p[GS][DS][A][B][C]$ can have ABC in any of the six permutation of $\bar{G}$, $\bar{D}$, and Z_p. We have implemented SweepKernel, LTimes and LPlusTimes specifically for each of these nestings, resulting in 18 unique kernels.

No deep optimizations have been performed, just basic OpenMP thread-level loops and use of the "restrict" keyword. The intent is to provide a "plain-vanilla" implementation in order to make the base implementations as easy to understand as possible, and to not inadvertently tie KRIPKE to non-portable architectural or language features.

5 PERFORMANCE RESULTS

We have generated multiple computational results to explore the performance of both KRIPKE and ARDRA. Our first results are used to characterize ARDRA's performance to ensure that KRIPKE includes all computationally intensive operations. Next, we use KRIPKE for simple on-node performance studies with and without OpenMP threading. Finally, we present MPI scaling to large numbers of cores, which demonstrate that KRIPKE is a good representation of ARDRA and that both codes reasonably match theoretical parallel performance models of sweep algorithms.

We have used two different computers at LLNL for these performance studies. The first computer is RZMERL, which is a linux cluster comprised of 2.6 GHz Intel Xeon Sandy Bridge processors. This computer has 16 processors per compute node and allows for two threads per core. We have used this computer to study on-node performance, and have focused on the performance implications of different loop nestings.

The second computer we used in our performance study is VULCAN, which is an IBM Blue Gene/Q computer. This computer is comprised of 1.6 GHz IBM PowerPC cores, with 16 cores per node, and four hardware threads per core. VULCAN has 24,576 nodes connected by a 5D Torus network. We used this computer for on-node performance studies as well as MPI scaling studies to large core counts. We will show results for KRIPKE and ARDRA from this computer to demonstrate that KRIPKE is a reasonable respresentation of ARDRA's performance.

5.1 ARDRA Performance Breakdown

In order for KRIPKE to be a good proxy for ARDRA, KRIPKE must include essential computational components that determine ARDRA's runtime. KRIPKE includes the LTimes, LPlusTimes, and Sweep components of ARDRA to approximate ARDRA's computational performance. Figure 1 shows ARDRA's runtime break down for each of these components on a weak scaling study from a single core to 32,768 cores on VULCAN. The top line in this figure is the total time for the Solve of

a simple k-eigenvalue problem. The line directly below the Solve time is the sum of the runtimes for LTimes, LPlusTimes, and the Sweep. The remaining lines are the runtimes for each individual component. This plot demonstrates that ARDRA's computational performance is dominated by these three solution components. On a single core, these three components account for 84% of the runtime; at large scale the three components account for 90-93% of the total Solve runtime. Furthermore, the difference between the total Solve time and the sum of the three components is constant as the problem is scaled from one core to 32,768 cores. This indicates that the computation represented in the other parts of the algorithm is not as important when studying ARDRA scaling. From these results, we conclude that KRIPKE contains the essential computational components to accurately represent ARDRA's computational performance.

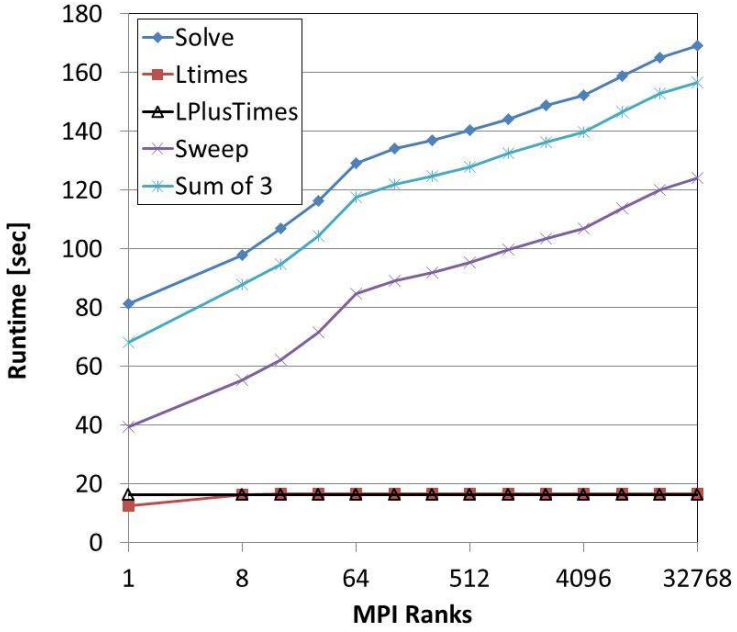


Figure 1. ARDRA timing breakdown

5.2 On-Node Performance Studies

We have defined a simple test problem for our on-node studies using KRIPKE. This problem has 64 energy groups, 96 total directions (12 per octant), P4 scattering, 12x12x12 spatial zones per core, and is run for 10 iterations.

Our first study investigates how the choice of loop nesting, group sets and direction sets affects the performance of this test problem on a single RZMERL node, running with one core per node. For this study, KRIPKE was built without OpenMP enabled. We used seven variations of group sets, six variations of direction sets, and all six loop nestings for a total of 252 code runs in the study. Table I summarizes these variations for this test problem. The first row in the table represents the best case for on-node performance, the last value in each column represents ARDRA's data layout.

Group Sets:Groups per set	Direction Sets:Directions per set	Loop Nesting
1:64	1:12	ZDG
2:32	2:6	ZGD
4:16	3:4	DGZ
8:8	4:3	DZG
16:4	6:2	GZD
32:2	12:1	GDZ
64:1		

Table I. Summary of On-Node variations

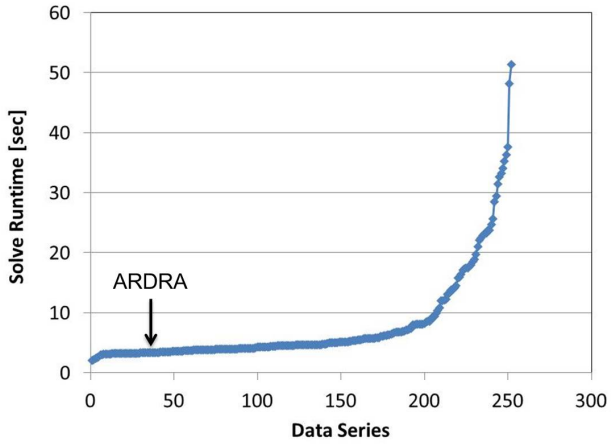
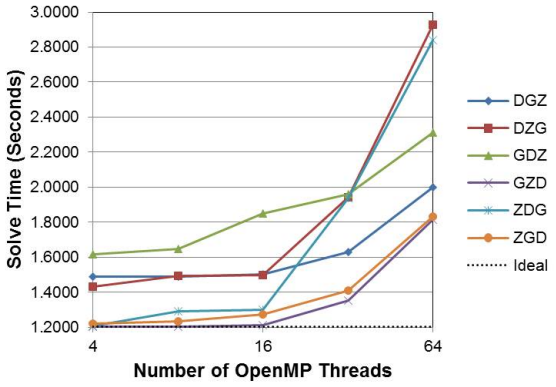


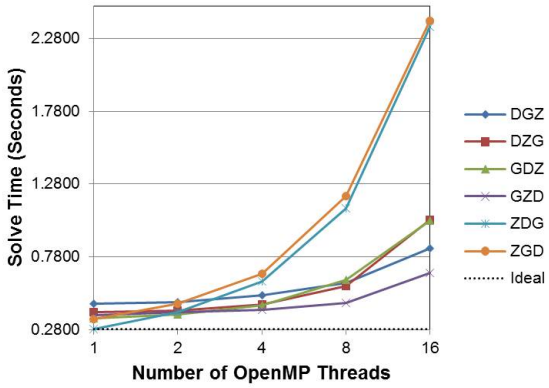
Figure 2. KRIPKE On-Node Performance for 252 variations

Results for the 252 runs are shown in Figure 2. The variation associated with ARDRA is shown on the plot. The fastest run is approximately 60% of the runtime compared to the ARDRA variation. The range in performance of all variations is 2 to 51 seconds. These results indicate that loop nesting and group and direction set choices will have a profound impact on performance.

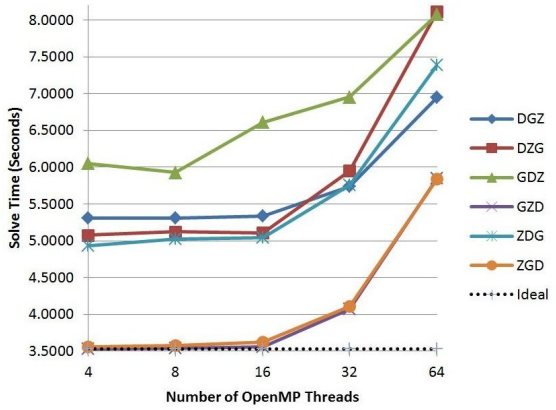
The second on-node performance study we ran with KRIPKE is designed to study the impact of OpenMP threads on on-node performance. For this study, we applied weak scaling by increasing the number of spatial zones as we increased the number of threads on a node. This study was run on RZMERL and VULCAN using the same parameters as the previous study, but also varying the scattering order from P4 to P9. The best results for each nesting are plotted. The P4 results for this study are found in Figures 3a and 3b, and P9 results in Figures 3c and 3d. If scaling with respect to increasing threads was perfect, the plots would be flat.



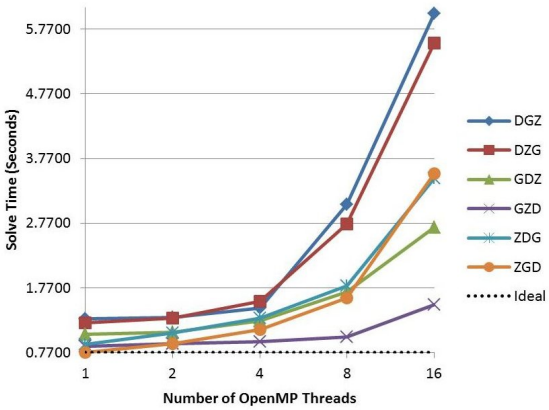
(a) KRIPKE, 1-Node, P4 Scattering, VULCAN



(b) KRIPKE, 1-Node, P4 Scattering, RZMERL



(c) KRIPKE, 1-Node, P9 Scattering, VULCAN



(d) KRIPKE, 1-Node, P9 Scattering, RZMERL

Figure 3. KRIPKE 1-Node P4 and P9 Scattering on RZMERL and RZVULCAN

The results from this study seem to show that the GZD scaling outperforms the others in most cases, however we need to study this further. In Figures 3b and 3d, we can see that the single thread performance is slightly better with ZDG and ZGD on RZMERL. On VULCAN with P9 scattering, ZGD and GZD are quite similar. We can see from this study that the relative performance of different nestings changes dramatically based on the computer architecture and the problem specification. Further research will be needed to determine if a single nesting choice is reasonable for the parent code, or if multiple nestings may need to be supported.

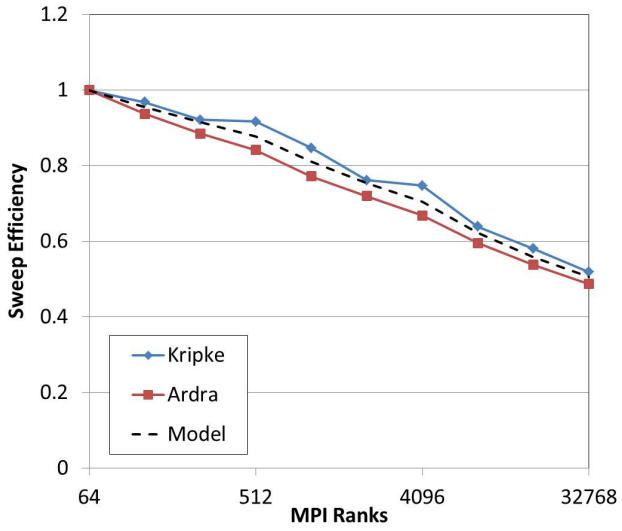
5.3 MPI Scaling and Comparison to ARDRA

We have run comparable weak scaling studies with KRIPKE and ARDRA to further explore KRIPKE's validity as a proxy for ARDRA. This weak scaling study has been run for both codes up to 32,768 cores on VULCAN. The scaling test problem used 48 groups, with S8, S12, and S16 quadrature sets, 12x12x12 spatial zones per core, and 10 iterations of the solves (10 full sweeps involving all groups and angles). This is the same scaling study performed in [14]. For KRIPKE we used the GDZ loop nesting, which is equivalent to ARDRA's implementation.

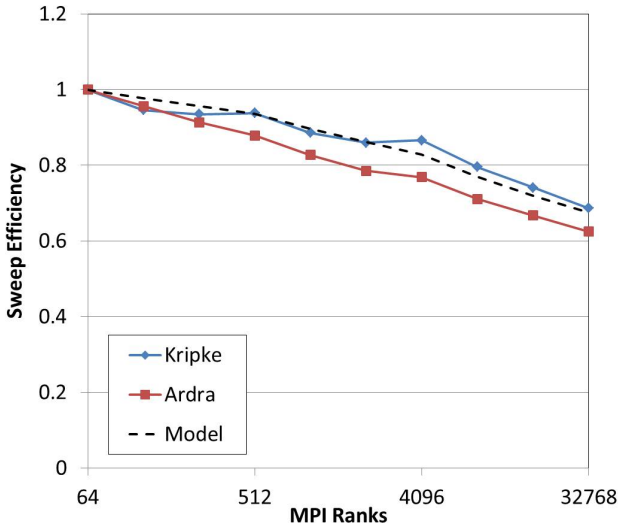
We show scaling plots for S8, S12, and S16 in Figures 4a, 4b, and 4c, respectively. In these figures, we plot efficiency, which is defined as

$$\epsilon = T_{sweep,P_{ref}}/T_{sweep,P}. \tag{6}$$

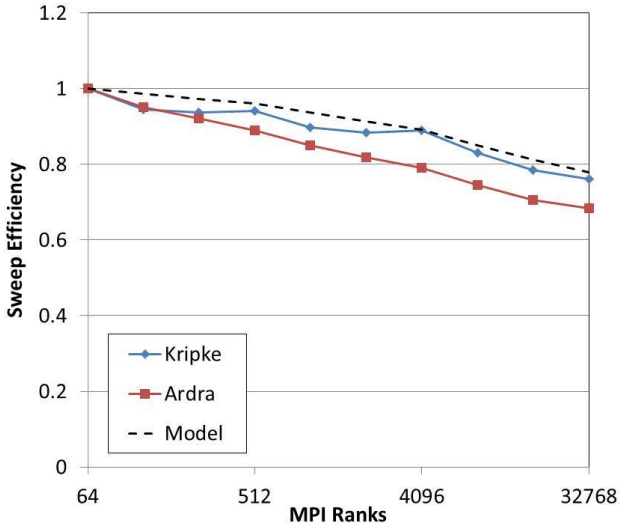
In all results we used 64 MPI ranks as the reference value for each weak scaling study. The model we used is consistent with the parallel performance models found in [15] and [13]. It is interesting to note that KRIPKE and ARDRA results tend to bracket the parallel performance model results. KRIPKE appears to scale a bit better, and is faster than ARDRA. This behavior is somewhat expected because KRIPKE only approximates the parallel algorithms in ARDRA. Possible reasons for the observed differences between the two codes include missing kernels and physics in KRIPKE, different memory layouts, and even compiler options. The general trends and performance are the same between the mini-app KRIPKE and its parent code ARDRA.



(a) S8 Weak Scaling results on VULCAN



(b) S12 Weak Scaling results on VULCAN



(c) S16 Weak Scaling results on VULCAN

Figure 4. S8, S12, S16 Weak Scaling results on VULCAN

6 CONCLUSIONS

We have shown that the performance characteristics of KRIPKE and ARDRA are similar, which demonstrates that KRIPKE is a good proxy application for ARDRA. This will allow us to use KRIPKE as a research tool, quickly iterating on new ideas, and potentially lead to improvements in ARDRA performance.

One of the initial goals for Kirpke was to learn what nesting would perform the best on different architectures, and explore the benefits of the GroupSet and DirectionSet concepts. The main takeaway from Figure 3 was that the choice of data layout is very dependent on problem specification, parallel scale, and architecture. There is no obvious choice as to which nesting is "best", which has led us to investigate supporting all 6 nestings in ARDRA. We are starting to learn from Kripke what kind of software engineering approaches may allow a production transport code to support multiple data layouts in a maintainable way. Also, it appears to be important to support GroupSet and DirectionSet concepts in order to improve performance and flexibility. However, it is ongoing research to more fully understand the performance implications for a given choice of GroupSet and DirectionSet sizes.

We have some short term goals that will enhance KRIPKE's flexibility and usefulness as a research tool. Ongoing research has shown that spatial domain overloading has the potential to greatly improve the scalability of the sweep algorithms in KRIPKE and ARDRA [14, 15]. Due to KRIPKE's simplicity, we can easily add this capability to the code and characterize the performance of general overloading algorithms on the parallel efficiency of sweep algorithms. We also would like to add different spatial discretizations, such as a discontinuous finite element discretization (DFEM), which would help us further characterize loop nesting requirements. DFEM's are much more computationally intensive than Diamond Difference, and have the potential to be more cost effective on new computer architectures.

We have a number of long term goals for KRIPKE as well. Adding more kernels to KRIPKE may be needed in order to reduce the differences between KRIPKE and ARDRA. Implementing unstructured grid sweep algorithms in KRIPKE will allow us to research the algorithms and software design required to successfully implement them in ARDRA. Exploring new programming models or languages in KRIPKE will allow us to tackle multiple data layouts and architectures, without having to maintain several variants of our codes and kernels.

KRIPKE has already shown its utility in studying a wide range of on-node performance issues, as well as large-scale studies. We anticipate that it will become even more useful as new computer architectures are being delivered to LLNL.

7 ACKNOWLEDGEMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

8 REFERENCES

[1] C. H. Still et al., "Estimating the Impact of Advanced Architectures on ASC Multiphysics Codes in the Absence of an Exascale Initiative," LLNL-TR-654332, Lawrence Livermore National Laboratory (2014).

[2] A. J. Kunen, "Kripke - User Manual v1.0," LLNL-SM-658558, Lawrence Livermore National Laboratory (2014).

[3] U. Hanebutte and P. N. Brown, "Ardra, Scalable Parallel Code System to Perform Neutron and Radiation Transport Calculations," UCRL-TB-132078, Lawrence Livermore National Laboratory (1999).

[4] M. R. Dorr and C. H. Still, "Concurrent Source Iteration in the Solution of Three-dimensional, Multigroup, Discrete Ordinates Neutron Transport," *Nucl. Sci. Eng.*, **122(3)**, pp. 287–308 (1996).

[5] E. E. Lewis and W. F. Miller, *Computational methods of Neutron Transport*, American Nuclear Society, La Grange Park, IL, USA (1993).

[6] P. N. Brown, "A Linear Algebraic Development of Diffusion Synthetic Acceleration for 3-D Transport Equations," , **32**, *1*, pp. 179–214 (1995).

[7] M. L. Adams and E. W. Larsen, "Fast iterative methods for discrete-ordinates particle transport calculations," *Prog. Nucl. Energy*, **40(1)**, pp. 3–159 (2002).

[8] G. G. Davidson, T. M. Evans, J. J. Jarrell, S. P. Hamilton, and T. M. Pandya, "Massively Parallel, Three Dimensional Transport Solutions for the *k*-Eigenvalue Problem," *Nucl. Sci. Eng.*, **177**, *2*, pp. 111–125 (2014).

[9] C. Clouse, "Parallel Deterministic Neutron Transport with AMR," *Proc. Computational Methods in Transport Workshop*, Tahoe City, CA, September 11-16, 2004.

[10] J. Reinders and J. Jeffers, *High Performance Parallelism Pearls*, Morgan Kaufmann (2014).

[11] OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 3.0," 2008.

[12] M. P. I. Forum, "MPI: A Message-Passing Interface Standard Version 3.0," 2012, Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

[13] M. P. Adams et al., "Provably Optimal Parallel Transport Sweeps on Regular Grids," *Proc. International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering*, Sun Valley, ID, May 5-9, 2013.

[14] W. D. Hawkins et al., "Validation of Full-Domain Massively Parallel Transport Sweep Algorithms," *Trans. Amer. Nucl. Soc.*, **111**, pp. 699–702 (2014).

[15] T. S. Bailey and R. D. Falgout, "Analysis of Massively Parallel Discrete-Ordinates Transport Sweep Algorithms with Collisions," *Proc. International Conference on Mathematics, Computational Methods & Reactor Physics*, Saratoga Springs, NY, May 3-7, 2009.